From Declarative Model to Solution: Scheduling Scenario Synthesis

Bruno Blašković, Mirko Randić
University of Zagreb
Faculty of Electrical Engineering and Computing
{$bruno.blaskovic, mirko.randic$}@$fer.hr$

***Abstract*—This paper presents deductive programming for scheduling scenario generation. Modeling for solution is achieved through program transformations. First, declarative model for scheduling problem domain is introduced. After that model is interpreted as scheduling domain language and as predicate transition Petri net. Generated reachability tree presents search space with solutions. At the end results are discussed and analyzed.**

I. Introduction

Two general directions are under consideration in this paper. First, deductive programming will be used as methodology for solution to scheduling problem. The second direction is experience that improves protocol synthesis due to synergism between scheduling and deductive programming. Declarative programming is concerned about *what* is to be done rather then *how* is it implemented. Declarative model is interpreted and transformed to executable model.

In this paper word *model* is frequently used.

Model can present requirements, program, agent or behavior in pure mathematical way or through the program code. Nowadays, there are numerous formal methods, specification languages, model checking and theorem proving tools. Putting together different methods, tools and languages is obtained through model transformation, component composition software composition or similar methods.

This paper use different models, each of them is suitable for its particular purpose. Declaration part comes from language specialized for scheduling problem definition. Executable part is found in high level Petri net. Together, by means of model transformation solution to the problem is found.

This paper is structured as follows: before model translation between declarative and executable models two solutions are presented, one by means of *Predicate Petri net* (Pr/T) in Section II and the other by *Planning Domain Definition Language* ($PDDL$) in Section III, respectively.

Working example is introduced in textual form in Section V. In Section IV unification between the Pr/T and $PDDL$ model yielding translation between the models is introduced. Section VI introduces metamodel as generalization of model transformations.

Experience from model translation and scheduling synthesis is used for scenario synthesis in Section VIII.

Solution to scheduling problem as *extended finite state machine* ($eFSM$) and $ITU-T$ *message sequence diagram* (MSC) is in Section VII.

Final Sections of the paper bring related work (Section X) with some reflection regarding synthesis process (in Section IX) as well as briefly recapitulate *literate programming* methodology and `noweb` tool.

At the end in Section XI is conclusion with further research directions.

II. Predicate Petri net

In this Section *Predicate Petri net* (Pr/T) solution is described. Textual problem from working example (Section V) is defined by (Pr/T) constructs and analyzed. In following text *working example* will be referenced as `4ws1tob-problem` shorter as `4ws1tob`.

From the modeling point of view two models can be identified:

- mathematical model: Pr/T is introduced as 6–tuple
- program code that is input to PrT tool for analysis

Predicate-Transition Petri net definition is taken from [7]. The tool implementing Pr/T [8] has been derived following the same formal definition. Pr/T is 6–tuple structure or mathematical Pr/T model (S, T, F, K, W, M_0) such that:

S is the set of places,

T is the set of transitions, $S \cap T = \emptyset$,

F is the set of arcs, $F \subseteq (S \times T) \cup (T \times S)$,
K is the capacity function, $K \in (S \to N_\omega)$,
W is the arc weight function, $W \in (F \to (N \setminus \{0\}))$,
M_0 is initial marking (in initial state) $M_0 \in \mathcal{M}$ where $\mathcal{M}$ is the set of markings (states), $\mathcal{M} = \{M \in (S \to N) \mid \forall s \in S \quad M(s) \leq K(s)\}$.

Pt/T tool used in this paper is PROD [8], [9]. Analysis is performed by means of reachability tree generation.

Figure 1. represents programming model Pr/T for `4ws1tob-problem` example:

- places (represented as circles) are sides of the "bridge",representing *Safe* and *Unsafe* part of the bridge.
- transitions (represented as boxes) are actions or events (*toSafe* and *toUnsafe*) denoting "crossings": e_S is event when (s_i, s_j) are crossing from *Unsafe* to *Safe*, and e_U is (s_k) crossing from *Safe* to *Unsafe*, respectively
- $< m_0, m_1, m_2, m_3 >$ are markings

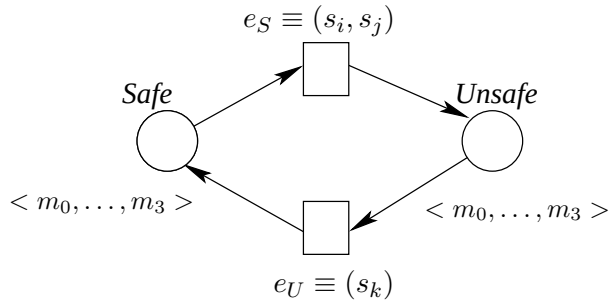

Fig. 1. Predicate Transition net for 4ws1tob problem (**Pr/T**)

There is no direct support for time in Pr/T as well as in PROD program. They are fulfilled afterwords (subsection II-A), by means of special program filter. More detailed description of Pr/T in $PROD$ syntax shows that Pr/T is also declarative `4ws1tob` problem description. In fact, graph structure from Fig.1 has program representation or program model that consists of:

(1) definitions: tokens in Pr/T are of type integers, they are used to "carry" information about elapsed time,
(2) places: *Safe* and *Unsafe*
(3) transitions *toUnsafe* and *toSafe*

Definitions are:

```
#define s1 10
#define s2 20
#define s3 25
#define torch 1
```

Unsafe place has initial markings describing "all soldiers are in *Unsafe* place":

```
#place Unsafe \
  mk
   (<.s0.>+<.s1.>+
    <.s2.>+<.s3.>+<.torch.>)
```

Each transition (*toSafe* and *toUnsafe*) implements previously mentioned events e_S and e_U, `in` is input place and `out` is output place, respectively:

```
#trans toSafe
in  {Unsafe:
       <.x.>+<.y.>+<.torch.>;}
out {Safe:
       <.x.>+<.y.>+<.torch.>;}


#trans toUnsafe
in  {Safe:   <.x.>+<.torch.>;}
out {Unsafe: <.x.>+<.torch.>;}
```

Goal is here expressed as computed tree logic (CTL) formula. Formula is used after reachability tree is generated. For that purpose separate program analyzer (`probe`) is used.

Safe place will eventually have all tokens (or *all soldier will be at safe side of the bridge*):

```
#define goal
EventuallyOnSomeBranch
(safe ==
 <.1.>+<.5.>+<.10.>+<.20.>+<.25.>)
```

All paths (branches in CTL PROD terminology) with solutions are present. In order to decrease reachability tree timing constraints are separately calculated.

A. Path filter: time analysis

Path filter selects only paths where goal–condition timing constraint holds:

$$t_{elapsed} = \sum_{i=1}^{m} t_i(e_i) \leq t_{max}$$

where:
$t_i(e_i)$ – event timing,
m – path length
One of such paths is presented in the Section VII.
As conclusion to this Section, experience from Pr/T analysis can be applied to scheduling scenario generation:

1) Pr/T has mathematical or formal model expressed as 6–tuple with program representation–model in `C-like` syntax denoted as $\mathcal{M}_{PrT}(pd = 4ws1tob)$
2) Pr/T is also declarative model because it describes structure, analysis through reachability analysis establish Pr/T as exe-

cutable model.Executable model is denoted as $\mathcal{M}_{PROD}(pd = 4ws1tob)$,

3) another declarative models that are established as a *n–tuple* consisting of entities, predicates, events/actions and similar structure can be transformed to Pr/T.

III. Planning Domain Description Language

PDDL (**P**lanning **D**omain **D**efinition **L**anguage) belongs to PDL (**P**roblem **D**omain **L**anguage) [6] class of languages. PDDL has syntax similar to Lisp and describes *what* has to be done rather than *how* is implemented. That fact makes PDDL natural candidate for declarative Modeling.

PDDL main purpose is to serve as input language for many planning tools. In this paper PDDL is used as declarative input whose syntax is more general and intuitive and that can hide formal method from the user [3].

Declarative PDDL model describing 4ws1tob-problem is 5–tuple:

$$\mathcal{M}_{PDDL}(pd = 4ws1tob) = (predicates, actions, objects, initial\_state, goal\_state)$$

where:

- objects: items of interest, for 4ws1tob objects are objects=$\{s_0, s_1, s_2, s_3\}$
- predicates: properties of objects, can be true or false, (example: Is s_i in state S ?)
- initial state(s): set of starting predicates formula (all s_i in *Unsafe*)
- goal state(s): set of goal predicates formula (all s_i in *Safe*)
- actions (operators): "crossing" the bridge expressed through *precondition* and *effect* predicates

In previous section (Sec.II) place and state are 'words' with similar but in general case different meaning, because PDDL and Pr/T languages have different semantic. In previous section (Sec.II) Pr/T has two models: mathematical and programming. PDDL has also two models, but both are expressed through *Lisp*-like syntax. PDDL can also serve as input to other planning and scheduling tools.

$\mathcal{M}_{PDDL}(pd = 4ws1tob)$ problem is expressed through PDDL constructs. Each construct is *Lisp*-like expression. Working example (4ws1tob from Sec. V) will be used to illustrate $\mathcal{M}_{PDDL}(pd = 4ws1tob)$ constructs. Now, we can say that PDDL program has the same syntax for mathematical and programming model. PDDL example starts with verbatim list of constructs:

```
(define (problem 4ws1tob01)
        (:domain 4ws1tob)
         (objects)
         (predicates)
         (initial_state)
         (goal_specification
         (actions_operators)))
```

Each construct will be described in more details.

A. Objects

Following notation from [10] types are introduced for each object:

a) s_0, s_1,s_2, s_3 are objects of *type* sold,
b) *torch* is object *type* torch,
c) *Safe*, *Unsafe* are objects of *type* place

Objects in PDDL are not object from object oriented programming paradigm. In *Lisp*-like syntax objects are defined by term rewriting:

```
(objects) ::==
   (:objects s0 s1 s2 s3 - sold
     torch - torch
     Safe Unsafe - place)
```

Now object construct is PDDL executable, that means planning tools can execute it. Similarly other constructs are rewrote (or replaced) producing declarative specification.

B. Predicates

Predicates can be used within other components. *Is token sold* s_i in *place* p_j ? *is expressed as*:

```
   (:predicates
        (pl ?sold ?place))
```

C. Initial states

In initial state component all tokens (s_0, s_1, s_2, s_3) are in *Safe* place and *Unsafe* place is empty. Timing parameters t_i are set, too. Initial time is set as:

```
(:init
 (= (t-elapsed) 0)
```

Initial state components are coded as follows: if token ?x is in place *Unsafe* than token ?x is not in place *Safe*, yielding following initial conditions:

```
 (pl s0  Unsafe)
 (not (pl s0  Safe)) (= (ts s0) 5)
 (pl s1  Unsafe)
 (not (pl s1  Safe)) (= (ts s1) 10)
 (pl s2  Unsafe)
 (not (pl s2  Safe)) (= (ts s2) 20)
 (pl s3  Unsafe)
 (not (pl s3  Safe)) (= (ts s3) 25)
 (pl torch Unsafe))
```

Predicate (= (ts s_i) t_i) initialize crossing time for object s_i.

D. Goal state

Goal specification component is theorem about system behavior. If solution exists place *Unsafe* is empty and all tokens of type *sold* are in place *Safe*. Solutions are found if goal is proved:

```
(:goal (and
  (pl s0 Safe)
    (not (pl s0 Unsafe))
  (pl s1 Safe)
    (not (pl s1 Unsafe))
  (pl s2 Safe)
    (not (pl s2 Unsafe))
  (pl s3 Safe)
    (not (pl s3 Unsafe))
  (pl torch Safe)
```

Goal has timing goal condition $t_{elapsed} \leq 60$ expressed as:

```
  ((<= t-elapsed) 60)))
```

E. Actions

Action operators realize the following functionality:

a) two objects (or tokens) are transfered from *Unsafe* to *Safe*, time $t_{elapsed}$ incremented
b) single object (or token) is transfered from *Safe* to *Unsafe*, time $t_{elapsed}$ incremented
c) redundant token *torch* is left in PDDL because implementation must support silent–moves (ϵ-actions)
d) parameters ?x and ?y are of type *sold*

Objects are used within PDDL terminology while tokens are used within Pr/T terminology. Model transformations unifies objects and tokens, they will be mixed and used as synonyms. Each action consist of *preconditions* and *effect*:

- *Effect* is e_S or e_U event mentioned earlier in Fig. 1 Sec.II.
- *Precondition* must hold in order an *effect* takes place.
- *Preconditions* for *toSafe* action are two tokens of type *sold* in place *Unsafe*.
- *Precondition* for *toUnsafe* action is token of type *sold* in place *Unsafe*.

1) toSafe action:

Event e_S is realized with *toSafe* action:

```
(:action toSafe :parameters (?x ?y)
 :precondition
  (and (pl ?x Unsafe) (not (pl ?x Safe))
     (pl ?y Unsafe) (not (pl ?y Safe))
     (pl torch Unsafe) (not (pl torch Safe)))
```

Effect should place chosen tokens in *Safe* place:

2) toUnsafe action:

Event e_U is realized with *toUnsafe* action:

```
:effect
 (and
  (pl ?x Safe)
    (not (pl ?x Unsafe))
  (pl ?y Safe)
    (not (pl ?y Unsafe))
```

and increment elapsed time $t_{elapsed}$:

```
 (+ (t-elapsed
    (max (ts ?x)(ts ?y))))))
```

toUnsafe action is similar to *toSafe* action, single token of type *sold* is going to *Safe* place and token ?x is removed from *Safe* place and put into the *Unsafe* place. *Precondition* with *effect* is semantically equivalent to condition-event or Place-transition in Petri nets. That enables smooth model transition to non-colored Petri nets.

```
(:action toUnsafe :parameters (?x)
:precondition
 (and
  (pl ?x Safe)    (not (pl ?x Unsafe))
  (pl torch Safe) (not (pl torch Unsafe)))

:effect
 (and
  (pl ?x Unsafe) (not (pl ?x Safe))
  (pl ?y Unsafe) (not (pl ?y Safe))
  (+ (t-elapsed (ts ?x)))))
```

PDDL described in this paper produces the same results with (lpg) planning software. Program is executable after minor adjustments through software provided by [6] project.
Parameters ($N = 4$, $K_S = 2$, $K_U = 1$, $t_{max} = 60$) are preserved through transformation from $\mathcal{M}_{PROD}(pd = 4ws1tob)$ to $\mathcal{M}_{PrT}(pd = 4ws1tob)$.

IV. Programming for solution

In this paper intention is to derive executable model from deductive or declarative model. Terms *deductive* and *declarative* are used as synonyms although from the formal point of view it is not the same.
Intention is to define model ($\mathcal{M}(pd = 4ws1tob)$) as executable without inventing yet another specialized Modeling or specification language. That opens possibilities for reasoning about the model properties and consequently introduces validation in early development phase.
This hypothetical C program becomes deductive program. Deductive or declarative program must have implicitly defined algorithm that should deduce only from declarations and predicates output results. Such

C program describes *What* is done rather than *How* is it done.
The same proposition holds for $\mathcal{M}(pd = 4ws1tob)$ PDDL and PrT models. We shall use shorter notation, $\mathcal{M}(pd)$ where pd is always $pd = 4ws1tob$. $\mathcal{M}(pd)$ is focused on *What* is to be done rather then *How* is it done. Natural candidates for the model $\mathcal{M}(pd)$ translation are Prototype Verification System (**PVS**) [10], term–rewriting systems and *Lisp* family of languages. Our solutions uses *Lisp* like languages. *Modeling for solution* effect is achieved through the following model transformations presented as commutative diagram in Fig.2. Such approach verifies proof–of–concept through model transformation experiments.
TR_j and TR_k are program transformation routine. In practical solution TR_j and TR_k will be realized through the metamodel concept: deductive will be interpreted through metamodel, metamodel is translated to executive model afterwords. In this paper direct model translation is used. Metamodel facilities are introduced in Section VI. Each transformation

$$\mathcal{M}(pd) \xrightarrow{TR_j} \mathcal{PDDL}_i \xrightarrow{TR_k} \mathcal{PN}_p$$

Fig. 2. Commutative diagram for model transformations

between models $\mathcal{M}(pd)$ require parser, because model transformation is program transformation. In order to avoid parser development following facts are considered:

1) mathematical models for PDDL is 5–tuple, introduced with lisp syntax,
2) mathematical models for PrT is 6–tuple, expressed as mathematical text, not as programming language
3) $PROD$ program is inC–like syntax and presents instantiation of PrT

PROD program describing PrT is coded in Lisp like constructs:

```
#trans toSafe
 in {Unsafe: <.x.>+<.y.>+<.torch.>;}
 out {Safe:  <.x.>+<.y.>+<.torch.>;}
```

becomes Lisp $PROD$ or $lPROD$:

```
(:trans toSafe
  :parameters (?x ?y ?torch)
     :in  (Unsafe ?x ?y ?torch)
     :out (Safe:   ?x ?y ?torch)
```

Note the similarity between PDDL `:action` construct and $lPROD$ `:trans` construct.

A. Unification

There is set of mappings between $PDDL$ and $lPROD$:
- : $init \longleftrightarrow$ initial-marking
- : $goal \longleftrightarrow$ final-marking
- : $action \longleftrightarrow$ `#trans`
- : $objects \longleftrightarrow$ `<.tokens.>`

Translation between $PDDL$ and $lPROD$ is straightforward: the set of mappings unify PDDL and $lPROD$.

V. Example scenario

This example belongs to the set of ”toy–problems” used in experiments during algorithm testing.

A. Textual scheduling problem definition

Working example is simple scheduling problem taken from [5], listed verbatim:

> Four soldiers who are heavily injured, try to flee to their home land. The enemy is chasing them and in the middle of the night they arrive at a bridge that spans a river which is the border between the two countries at war. The bridge has been damaged and can only carry two soldiers at a time. Furthermore, several land mines have been placed on the bridge and a torch is needed to sidestep all the mines. The enemy is on their tail, so the soldiers know that they have only 60 minutes to cross the bridge. The soldiers only have a single torch and they are not equally injured. The following table lists the crossing times (one-way!) for each of the soldiers:
> - soldier S_0 5 minutes
> - soldier S_1 10 minutes
> - soldier S_2 20 minutes
> - soldier S_3 25 minutes
>
> Does a schedule exist which gets all four soldiers to the safe side within 60 minutes?

VI. Scheduling domain metamodel

abstract model here – no metamodel needed here Scheduling domain metamodel is derived from PDDL model. Metamodel supports constructs from *type* theory, concurrency theory as well as process algebras.

After the analysis of the text from Section V the following list of `constructs` are introduced: parameters, entities, predicates, events, traces, initial–conditions, goal–conditions, operators and constraints.

In the next step each construct is described through *Lisp*-like constructs:

```
(def-abstract-semantic-net
    "metametamodel"
 (problem-domain 4ws1tob)
  (parameters construct)
  (entities construct)
  (predicates construct)
  (events construct)
  (traces construct)
  (initial-conditions construct)
  (goal-conditions construct)
  (operators construct)
  (constraints construct))
```

Parameters are data of types *integer* or *real*:

(1) number of soldiers $n = 4$
(2) ...`carry two soldiers` (to *safe* side) $K_S = 2$, and $K_U = 1$ (to *unsafe*) side.
(3) ...`cross times`: $t_0 = 5$, $t_1 = 10$, $t_2 = 20$, $t_3 = 25$,
(4) ...have only `60 min.` to cross the bridge

Torch is not considered here because it has not influence on model behavior. Next models ($\mathcal{PDDL}$, $hl\mathcal{PN}$) can include it but that is not necessary.

```
(def-parameters
 (n 4 int)
 (KS 2 int)
 (KU 1 int)
 (t0 5 real)
 (t1 10 real)
 (t3 20 real)
 (t4 25 real)
 (t-max 60 real))
```

Entities are two sets: one set are variables and the other are values. Set A is describing dynamic behavior of the model: in each execution step values from set A_S are assigned to variables from S.

1) set of soldiers: $s_0 \dots s_3$ of *type* `sold`
2) set describing sides of the bridge. They are introduced as *places* (*Safe* side and *Unsafe* side) of *type* `place`.

Unknown parameter is denoted as $?A$ or $?AS$.

```
(entity (A (s0 s1 s2 s3 s4))
(entity (AS (Safe Unsafe))
```

Predicates answers the question:

(1) Where is s_i ?
(2) Is s_i in side *Safe* or *Unsafe*

```
(pred atPlace ?A)
(pred ?A ?AS)
```

There are two atomic events:

(1) two soldiers s_i and s_j are going to *Safe* side
(2) single soldier s_i is going to *Unsafe* side
(3) event has duration time t_i

```
(eS (Unsafe (?x ?y) Safe)
    (time (max (?tx ?ty))))

(eU (Safe ?x Unsafe)
    (time (?tx )))
```

If there is solution for this problem traces should be of finite length r coded as finite length vector, such that `?e` is `?eS` or `?eU` event of duration `?total-time`. This model has no built–in infinite traces.

```
(Er (foreach r ?e) ?total-time)
```

Initially all s_i are on *Unsafe* place. This model has no time counter (initial-condition (A (atUnsafe atUnsafe atUnsafe atUnsafe))) At the end all soldiers must be within 60 minutes in safe side:

```
(goal-condition
   (A (atSafe atSafe atSafe atSafe))
   (<= total-time t-max))
```

Operators and *constraints* constructs serve as additional model input in complex situations where $\mathcal{M}(pd)$ is profiled recursively.

VII. Solution

There are 16 paths from total of 824 paths where timing condition $t_{elapsed} \leq 60\,min$ holds. As an example one path is presented:

```
PATH 33
Node 0: transition toSafe
  x = 5  y = 10
Node 1: transition toUnsafe
  x = 5
Node 7: transition toSafe
  x = 25  y = 20
Node 13: transition toUnsafe
  x = 10
Node 20: transition toSafe
  x = 10  y = 5
Node 21
```

`Node 0,1,5,13` ...are nodes from reachability tree. Variable x and y are crossing times, for *toSafe* transition or e_S event crossing time is $max(x, y)$.

A. Visualization: $e\mathcal{FSM}$

Fig.3 visualize [14] solution in the form of extended Finite State Machine (`eFSM`). Next step can transform `eFSM` into the input language for analysis tool. Another possibility is to generate skeleton code in `C` or `java` programs.

B. MSC solution

Message sequence charts [17] is another form that can visualize solution (Fig.4). Even the

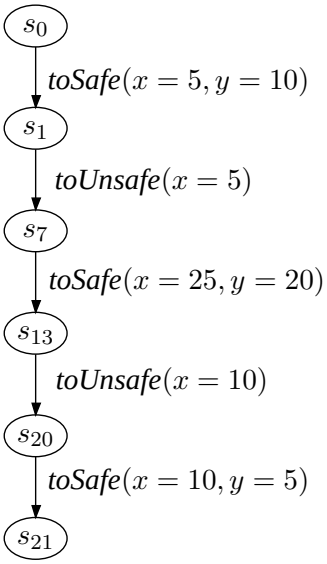

Fig. 3. solution as real–time program (*eFSM*)

more MSC can be used as source for another set of translations into the statecharts, SDL diagrams ...

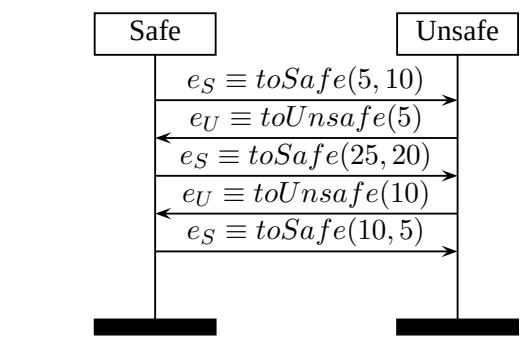

Fig. 4. solution as Message Sequence Diagram(**MSC**)

VIII. Synthesis and scheduling

How can *planning* and *scheduling* methods influence synthesis?
Synergy effect between scheduling and synthesis opens possibilities for interpreting (`eFSM`) in various ways. Each of them benefits through skeleton or templates for `code`, `scripts` or `architecture` definition. During requirement phase many different scenarios can be automatically generated. We found that "side–effects" or parts of scenarios–specifications that are less obvious but present are reduced.
Decidability and computability of our approach is not optimal for big examples because of state explosion problem. Heuristic scheduling algorithms are of little practical importance for synthesis problem. Different problem will in most cases have different declarative program. For that purpose Petri net reachability algorithms will be replaced with satisfiability (SAT) algorithms.
Some interpretation of solutions are:

a) protocol synthesis
b) `SDL` process: skeleton of program SDL can be generated and used by designer
c) `MSC` skeleton processes can be composed in system. Overall behavior is analyzed as early as in requirement phase
d) parallel program job scheduling: an experiment for dynamic job allocation for parallel program is planned using proposed methodology

Besides mentioned interpretations other possibilities are:

a) real–time system job scheduling
b) control software synchronization
c) performance prediction
d) ontology definition concept analysis
e) system maintenance
f) object and methods definition and optimization

Proposed approach introduces methodology for identifying and minimization of states in `FSM` like models of concurrent reactive systems.

IX. Scope and motivation

There are various approaches for synthesis problem, the most significant are:

(a) the temporal logic formula describes the system. Synchronization part of the system is derived from temporal formula. This can be, roughly speaking, interpreted as reverse model checking.
(b) from formal service specification to protocol specification. Formal description is transformed into protocol specification. Even the more, in most cases specification is executable enabling verification, simulation and analysis ...
(c) extended finite state machine is constructed from executable traces. Traces are sequences of messages, signals or sequence of events. They can be defined by designer, in this paper intention is to provide traces automatically to the designer.

Our approach introduce traces or more preciously event traces as declaration for desired system behavior. An event describe *crossing the bridge* (example from Section V. Sequence of events define trace. The set of all traces represents search space where

solution should be found taking timing constraints into the considerations.
In this paper traces are interpreted as Petri net reachability tree paths. Reachability tree paths and traces describe the same behavior model.
If various models $\mathcal{M}$ have same behavior model then they can be transformed. Transformation is mapping of constructs between models, before mapping constructs are unified.
Only paths with desired property (crossing time limit) are solution paths.
Another question is how to only generate traces that are solution i.e. to avoid state–space combinatorial explosion. Declarative meta–model has no knowledge about it.
Modeling for solution has three steps:

(i) model definition
(ii) translation to domain specific language (in our case `PDDL - scheduling&planning` language). `PDDL` can be used as input for scheduling planning tools.
(iii) translation to high level Petri net for analysis and solution finding.

It is obvious fact that *model–for–solution* can start and find solution from step *(ii)* or step *(iii)* without the model. In complex situations, when system is not formally described, where constraints and assertions about the system are contradictory, unknown, unclear or unspecified such model–mixing proves its value. Another motivation is to give designer or modeler support to–play–with with different tools and approaches in order to achieve desired quality of solution. Formal approaches explore the benefits and experience from automatic deductive programming, program transformation as well as literate programming.
Previous work were focused on synthesis as component composition: smaller architectural parts or system blocks were composed into the target system. Such approach has usable results for component based architectures like services definition within the intelligent networks as found in numerous $ITU-T$ recommendations. Later on, working example problem is solved with high level Petri net in a way close to approach described in Sec.II. As a consequence, scheduling scenario interpreted as MSC scenario yields another synthesis approach. Such interpretation can produce MSC chart as solution or synthesize executable specification by means of scheduling methodology. Modeling for solution follows experiments towards synthesis of scenarios and its translation to finite state machine based systems like statecharts or ITU-T SDL language. This paper also try to address such question through reachability tree analysis of scheduling solver. Results and methodology from another research field (planning and scheduling) are exercised, yielding synergistic effect on protocol or concurrent reactive system design. Experience shows that scheduling problem generalization and synthesis issues can benefit from each other.
In [19] Modeling framework suitable for experiments is introduced. Modeling framework consists of several levels, each level describes position in model hierarchy, from the most abstract level on the top to implementation level at the bottom. Within each level components are introduced (traditionally called $\mathcal{ECP}$ (**E**lementary **C**ommunicating **P**rocesses) as black boxes that enables program, tools or even models inter working.

X. Related work

In [16] synthesis is described as message sequence chart (MSC) translation into the Real-time Object Oriented Model (ROOM). After that, designer can use ROOM model for simulation as well as other purposes. Formals description technique (MSC) describing system architecture and behavior is interpreted as executable model. MSC serves as *top–level–model* which can be analyzed, simulated and implemented.
Functional specification of the problem and temporal logic yields state–based automaton as solution for elevator problem [15]. Satisfiability analysis generates synchronization part of the system.
Synthesis of behavior models from scenarios is introduced in [1] and [2].
`PROMELA` model serves as input of `spin` protocol verifier from [5]. Results are presented through MSC diagrams.
Results from mathematical description with process algebra and concurrency presented in [4] are used for further development of metamodel described in Section VI. Another metamodel comes from [18]. Model transformation routines are developed by means of `noweb` literate programming tool. Literate programing discipline has been introduced by D.E. Knuth *...instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*
There are many literate programming supporting tools [13] providing human readable files that incorporate documentation and source code into the single file. In this paper all sections illustrating

concepts and constructs (functional style programs) are produced with literate programming tool noweb [12], [11].

XI. Conclusion and further work

Synergy effect between planning, scheduling and synthesis can improve design process. There are no universal approach for synthesis problem so only narrow problem domains are possible to solve with difficulties regarding NP-hard algorithms and undecidable problems. This papers describe proof–of–concept rather then industrial strength approach.

Pros (+) and cons (-) can be summarized as follows:

(+) synergism between synthesis and scheduling planning: all ready developed routines for scheduling have been adopted and used

(-) state explosion: Petri net can produce unmanageable reachability tree size. Reachability analysis tool support is designed for model checking. Some scheduling issues are unsuitable for model–checking technology

(-) narrow problem domain: declarative model requires significant changes with small domain change

(-) small scale problems: synthesized components are sometimes easier to handle by hand

(+) proof of concept: model transformation is usable programming paradigm

(-) complex theoretical background: designer should have deep understanding of all models and translation process

(+) solution for critical applications: mission critical software can be developed in this way yielding stable solutions

(+) open research platform: modifications and updating to new algorithms

(+) interworking of different paradigms and formal methods

Further work will (1) use satisfiability algorithms and (2) explore formal methods interworking. Synthesis method should serve as testbed for formal languages semantic analysis and executable specification languages.

References

[1] S. Uchitel and J. Kramer, "A workbench for synthesising behaviour models from scenarios," in *Proceedings of the* 23^{rd} *Intl. Conf. on Software Engineering*. Toronto, Canada: ACM press, 2001.

[2] S. Uchitel, R. Chatley, J. Kramer, and J. Magee, "System architecture: the context for scenario-based model synthesis," in *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 2004, pp. 33–42.

[3] J. Rushby, "Disappearing formal methods," in *High-Assurance Systems Engineering Symposium*. Albuquerque, NM: Association for Computing Machinery, nov 2000, pp. 95–96. [Online]. Available: http://www.csl.sri.com/papers/hase00/

[4] B. Bruno and M. Randic, "Model based scheduling scenario generation," in *DAAAM International Scientific Book 2006* (B. Katalinic, ed.), ch. 04, pp. 031–044, DAAAM International Vienna: TU Wien, Karlsplatz 13/311, A-1040 Vienna, Austria, 2006.

[5] T. C. Ruys and E. Brinksma, "Experience with Literate Programming in the Modelling and Validation of Systems," in *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)* (B. Steffen, ed.), no. 1384 in Lecture Notes in Computer Science (LNCS), (Lisbon, Portugal), pp. 393–408, Springer-Verlag, Berlin, Apr. 1998.

[6] A. Gerevini and D. Long, "Plan constraints and preferences in pddl3," tech. rep., Dept. of Electronics for Automation, University of Brescia, Italy, Technical Report, RT 2005-08-47, 2005.

[7] K. Varpaaniemi, "Efficient Detection of Deadlocks in Petri Nets," Helsinki University of Technology, ESPOO, Finland, Series A: Research Reports 26, October 1993.

[8] P. Grönberg, M. Tiusanen, and K. Varpaaniemi, "PROD – A Pr/T–Net Reachability Analysis Tool," Series B: Technical Reports 11, Helsinki University of Technology, ESPOO, Finland, June 1993.

[9] K. Varpaaniemi, J. Halme, K. Heikkanen, and T. Pyssysalo, "PROD Reference Manual," Series B: Technical Reports 13, Helsinki University of Technology, ESPOO, Finland, August 1995.

[10] S. Owre and N. Shankar, "Abstract datatypes in PVS," Tech. Rep. SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec 1993. Extensively revised June 1997; Also available as NASA Contractor Report CR-97-206264.

[11] A. L. Johnson and B. C. Johnson, "Literate programming using *noweb*," *Linux Journal*, pp. 64–99, October 1997.

[12] N. Ramsey, "Literate programming simplified," *IEEE Software*, vol. 11, pp. 97–105, Sept. 1994.

[13] M. Smith, "Towards modern literate programming," Project Report HONS 10/01, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, 2001.

[14] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software — Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[15] A. Ursu, V. Dubenetsky, and G. Gruita, "Design of real time systems using temporal logic specifications: a case study," *Computer Science Journal of Moldova*, vol. 5, no. 2, pp. 88–114, 1997.

[16] S. Leue, L. Mehrmann, and M. Rezai, "Synthesizing ROOM Models from Message Sequence Chart Specifications," Tech. Rep. 98–06, Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, April 1998.

[17] ITU, *Z.120 Message Sequence Charts (MSC96)*, ITU-T Telecommunication Standardisation Sector, Geneva, 1996.

[18] Frédéric Jouault and Jean Bézivin, "KM3: a DSL for Metamodel Specification," pp. 171–185, 2006. [Online]. Available: http://www.lina.sciences.univ-nantes.fr/Publications/2006/JB06a

[19] Bruno Blašković , "Petri Net Modeling for Reactive System Verification," in *Proceedings of 7th ConTEL*, vol. (1), (Zagreb), pp. 257–264, FER–IEEE, June 2003.